SPOOL: A simple pseudo object oriented language

Author: Muraru Liviu

University "Alexandru Ioan-Cuza" Iași, Faculty of Computer Science

Abstract. This paper contains documentation for the author's final project on the course of **"The principles of programming languages"** for the 2017-2018 academic year of study, taught at the institution mentioned above.

1 Introduction

The following sections provide a definition for ***SPOOL***(which the author chose as a starting point in creating a hand-made parser) and should be considered a proof of concept. The main subjects this paper covers are:

- the extended-BNF syntax of ***SPOOL***
- examples of textual programs using ***SPOOL***
- parsing ***SPOOL***
- what could be done to improve ***SPOOL*** and/or the parser
- the starting ideas for the parser

Note that there is no section that includes the semantics of ***SPOOL***. That is because ***SPOOL*** is not supposed to be the main subject of this paper, but a starting point in creating a parser.

2 Syntax

Below you will find the **extended-BNF** definition of ***SPOOL***'s syntax:

⟨*expr*⟩ := Integer | Id | ⟨*expr*⟩ '+' ⟨*expr*⟩ | ⟨*expr*⟩ '-' ⟨*expr*⟩
| ⟨*expr*⟩ '*' ⟨*expr*⟩ | ⟨*expr*⟩ '/' ⟨*expr*⟩
| ⟨*expr*⟩ '%' ⟨*expr*⟩ | '(' ⟨*expr*⟩ ')'
| 'create' S:Id | 'self'
| X:Id '=' ⟨*expr*⟩
| X:Id 'as type' B:Id

⟨*stmt*⟩ := ⟨*var_decl*⟩ ';' | 'break ;' | 'continue ;'
| 'if' ⟨*expr*⟩ 'then' ⟨*block*⟩ 'endif'
| 'if' ⟨*expr*⟩ 'then' ⟨*block*⟩ 'else' ⟨*block*⟩ 'endif'
| 'while' ⟨*expr*⟩ 'do' ⟨*block*⟩ 'endloop'

| ‘for’ X:Id ‘from’ ⟨*expr*⟩ ‘to’ ⟨*expr*⟩ ‘do’ ⟨*block*⟩ ‘endloop’
| ‘return’ (X:Id)$^{?}$ ‘;’ | ‘panic!(S:String);’

⟨*block*⟩ := ‘{’ ⟨*statement*⟩* ‘}’

⟨*var_decl*⟩ := ‘var’ *List*⟨*Id, ,*⟩

⟨*fn_def*⟩ := ‘fn’ X:Id ‘(’ ⟨*var_decl*⟩$^{?}$ ‘)’ ⟨*block*⟩

⟨*class_def*⟩ := ‘class’ T:Id ‘{’ ⟨*in_class_def*⟩* ‘}’
| ‘class’ T:Id ‘inherit’ T$'$:Id ‘{’ ⟨*in_class_def*⟩* ‘}’

⟨*in_class_def*⟩ := ⟨*fn_def*⟩ | ⟨*var_decl*⟩

⟨*type_def*⟩ := ⟨*class_def*⟩ | ⟨*fn_def*⟩

⟨*entry_point*⟩ := ‘entry’ ⟨*block*⟩ ‘exit’

⟨*program*⟩ := ⟨*type_def*⟩* ⟨*entry_point*⟩

The above grammar is a minimal set of production rules that defines a Turing-complete programming language(that is, it allows iterative and conditional constructs, and it provides a memory band). The grammar is subject to change.

3 Parsing

First and foremost, the reader should understand the **Parsing Problem**:
Input: A textual representation of a program written in **SPOOL**.
Output: A parse tree of the input, if it represents a valid program in **SPOOL**.

The decision of a parsing algorithm is premature, so I shall only predict a bottoms-up LALR(1)/LR(1)/SLR(1) parser will be utilized. For a few examples of **SPOOL** programs, refer to the files this paper was obtained with.

4 Conclusion and Outlook

There is plenty that could be improved with both **SPOOL** and the parser, so here's a most-immediate list:

- implement class level static memory in **SPOOL**
- implement function level static memory in **SPOOL**
- modularize the grammar to make it more scalable and sound(i.e. find a way to disable using **self** outside a class)
- create specialized trait/interface mechanism(yeah, sure...)
- implement access levels for class data